

JHDL - An HDL for Reconfigurable Systems *

Peter Bellows and Brad Hutchings †

Department of Electrical and Computer Engineering
Brigham Young University, Provo, UT 84602
bellows@ee.byu.edu, hutch@ee.byu.edu

Abstract

JHDL is a design tool for reconfigurable systems that allows designers to express circuit organizations that dynamically change over time in a natural way, using only standard programming abstractions found in object-oriented languages. JHDL manages FPGA resources in a manner that is similar to the way object-oriented languages manage memory: circuits are treated as distinct objects and a circuit is configured onto a configurable computing machine (CCM) by invoking its constructor, effectively “constructing” an instance of the circuit onto the reconfigurable platform just as object instances are allocated in memory with conventional object-oriented languages. This approach of using object constructors/destructors to control the circuit lifetime on a CCM is a powerful technique that naturally leads to a dual simulation/execution environment where a designer can easily switch between either software simulation or hardware execution on a CCM with a single application description. Moreover, JHDL supports dual hardware/software execution; parts of the application described using JHDL circuit constructs can be executed on the CCM while the remainder of the application—the GUI for example—can run on the CCM host. Based on an existing programming language (Java), JHDL requires no language extensions and can be used with any standard Java 1.1 distribution.

1 Introduction

When developing applications for configurable or FPGA-based computing machines (CCM), designers must perform two general tasks. First, they must design the circuitry that implements the necessary functionality for the

application. This is typically done using commercial CAD tools such as VHDL synthesis or schematic capture in concert with the back-end tools obtained from the FPGA device vendors. Second, designers must write a supervisory program that controls the configurable-computing platform during the operation of the application. In some cases this control program is relatively simple, just loading a single configuration and then loading and retrieving data. In more complex run-time reconfigured applications for example, these control programs can be relatively complex, loading a variety of configurations and data, on demand, as the application proceeds. Currently, the control program and the circuit description must be developed and simulated independently; the designer is responsible for ensuring that these two pieces of software cooperate correctly, typically through repeated download, execute and compile cycles on the CCM.

This division between circuit description and control program is really just a division of the application into its constituent static and dynamic parts: the static part represented by a circuit library, and the dynamic part embodied in a control program that chooses circuit configurations from a library, configures devices, and executes the application. However, given that dynamically changing hardware is at the core of configurable computing, treating the dynamic and static parts of the application independently is awkward and limiting. What is needed is a single integrated description that allows the designer to naturally express the dynamic and static parts of the application simultaneously. This paper describes a design approach/CAD tool that focuses on the creation of such an integrated description. As the number of Hardware Description Languages (HDLs) is innumerable and as it is difficult and time consuming to come up with pithy acronyms, we have chosen to name this system, JHDL, for Just another HDL.

2 Project Goals

The primary objective of this research project is to develop a tool-suite/design-environment for describing cir-

*Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) under contract number DABT63-96-C-0047. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

†Brad Hutchings is currently on sabbatical leave at Hewlett Packard Laboratories, Ltd, AppliCance Computing Department, Filton Road, Stoke Gifford, Bristol, UK BS12 6QZ

cuit organizations that dynamically change their structure over time. This project has the following additional requirements and potential benefits.

1. *It must use an existing programming language with no extensions.* This will make the tool accessible to a wider range of programmers by allowing them to use commercially-available compilers.
2. *The CCM-control paradigm must be CCM independent.* CCM control details should be abstracted to a higher level programming abstraction. This will make CCMs more accessible to programmers and will also ease the process of retargeting an applications to run on a variety of different CCMs.
3. *The description method must support run-time and partial configuration.* These are the most demanding CCM applications and will be used from the outset to stress the design environment.
4. *The integrated description must serve for both simulation and final execution with no modifications.* For simulation, it must support end-to-end simulation of applications that may consist of many configurations. For execution, it should be possible to switch transparently from a software simulation to hardware execution on the CCM simply by changing a software switch.

3 Background

There have been several efforts to create textual CAD tools for FPGA designs. In an early pioneering effort at DEC PRL, Vuillemin and his group developed and used Perle [9] to design CCM applications on DECPerle-1 and more recently on the Pamette [7]. Perle is a C++-based CAD tool that uses hierarchy and inheritance to describe user circuits. A Perle description, when compiled and executed, generates a netlist that is then processed by Xilinx place and route tools. Other similar examples of object-oriented circuit-design languages include Spyder [4] and Lola [2].

Run-time reconfiguration (RTR) has been receiving more attention lately and a few efforts are starting to report results with tools and run-time environments. Luk and Shirazi [5] reported on compilation tools for RTR designs. Their tools consist of a partial evaluator, an incremental configuration calculator and an optimizer. One of their goals is to automatically generate circuit overlays that have been optimized for use in partially configured applications. Burns and Donlin [1] reported on a run-time system for dynamic configuration. They proposed

a run-time system that attempts to automatically manage FPGA resources similar to the way a conventional OS manages memory or CPU resources. The system as proposed consists of a virtual hardware manager (for managing the FPGA resources), a transform manager (for modifying circuits to accommodate available device resources), a configuration manager (to manage the configuration process), and a device driver. Gokhale and Gomersoll [3] reported on their high-level compilation tools for fine-grained FPGAs such as the National CLAY device. These tools accept a dbC (data-parallel C) version of the algorithm, partition it into control and datapath and then implement the circuit using parameterizable module generators that have been optimized for fine-grained FPGAs. Lysaght has also reported on a VHDL-based simulation environment for RTR [6].

JHDL has some things in common with many of these efforts. First, as a design tool, it has been designed to directly support run-time reconfiguration, both partial and global, and it attempts to hide details of configuration from the user. However, in contrast to other work, JHDL makes no attempt to automatically identify partial configurations nor does it address the run-time physical transformation of circuits so that they will fit within available FPGA resources. At present, JHDL is primarily a manual design tool that combines CCM control and circuit design into single integrated description. JHDL probably has more in common with Perle as it uses hierarchy in a manner similar to Perle. However, it differs from Perle in that it was specifically designed to support run-time reconfiguration and CCM control.

Note that Java is not critical to this project; almost any object-oriented language would have sufficed. Java does have some useful features that can be exploited for this project, in particular, the portability and integrated GUI API are useful, however, any language that supports object construction and hierarchy would be a likely candidate for this project.

4 Research Approach

The primary distinction of JHDL and indeed the primary goal of this project is the creation of a single integrated API that allows the designer to express circuit organizations that dynamically change over time. Stated another way, the primary goal is to allow the designer to specify, in a reasonably natural way, when hardware gets loaded and removed from a CCM without exposing any of the details normally associated with CCM operation. Rather than invent a new language feature to schedule the configuration of circuits, we chose to adopt the object-instance construction/destruction mechanism used in object-oriented

languages. Conventional object-oriented languages manage memory through object constructors and destructors. Memory is allocated by invoking an object constructor that allocates the necessary memory from the heap or stack and sets object variables to initial values. Memory is reclaimed by invoking an object destructor that frees the memory back up to be used by other objects. JHDL manages FPGA resources on CCMs in a similar manner. In JDHL, all circuits are developed hierarchically as distinct objects. Allocating FPGA resources, i.e., configuring the FPGA devices, is performed by invoking the constructor for a circuit object and analogously, FPGA circuitry is reclaimed by invoking the circuit's destructor.

This approach of using object constructors/destructors to control the circuit lifetime on a CCM is a powerful technique that naturally leads to a dual simulation/execution environment where a designer can easily switch between either software simulation or hardware execution on a CCM with a single application description. When simulating in software, the constructors/destructors communicate with the JHDL simulation kernel. Constructors create object instances in system memory; these object instances are actually simulation models that interface with a simulation kernel to provide a clock-by-clock simulation of the user circuit. However, when executing in hardware (on the CCM), the constructors/destructors communicate directly with the CCM (through a JHDL interface layer) instead of the simulation kernel. Instead of allocating system memory, constructors load circuit descriptions from a circuit library and control the execution of the CCM. Analogously, destructors remove circuits by replacing existing circuits with "blank" configurations, similar to the state that exists when the FPGA is initially reset.

5 Overview

The system described in this paper is very much an experiment in progress. In these early stages, the main goal was to demonstrate feasibility of the constructor/destructor mechanism as a means for controlling a CCM; feasibility of object-oriented languages as circuit design tools has already been demonstrated by others. In its current state JHDL implements a circuit simulator and the control API for the Hotworks board from Virtual Computer Corporation (VCC). Netlisting capability is supported by an internal circuit graph data structure (hereafter referred to as the circuit graph) that is maintained automatically by the JHDL environment as circuits are constructed or destructed; however, an actual netlist format has not been determined at this point. (Note to reviewers: by the time of FCCM, primitive netlisting will be implemented.) As

such, all circuits for the ATR demonstration system discussed in this paper (the ATR shapesum circuits, for example) were designed manually using schematic capture with their matching simulation models written using the JHDL primitives. In addition, because this application uses runtime configuration with partial configuration of the 6200 devices on the Hotworks board, most circuits were hand placed. Full simulation of the ATR application was performed in JHDL and the results of the simulation completely match the hardware execution on the Hotworks CCM. Note that the JHDL circuit description serves as the *sole* means of controlling the VCC CCM, controlling all I/O and configuration sequencing.

6 The JHDL System

The current JHDL system is implemented as a set of Java class libraries with functionality divided into two basic areas: circuit simulation and CCM runtime support. Circuit simulation classes allow the designer to design circuit models that can be simulated at the clock level through the JHDL simulation kernel. CCM runtime support classes provide transparent access to CCM control functions via the construction/destruction mechanism described earlier.

Designers develop circuits in JHDL by selecting from a set of synchronous and combinational elements and wiring these together to form any arbitrary synchronous circuit. There are three different classes that can be used to implement a circuit: *CL* (combinational), *Synchronous* (clocked), and *Structural* (interconnection of combinational or synchronous elements). When creating a new circuit, the designer decides whether the outputs of the circuit are updated continuously, i.e., it is a combinational circuit (a *CL* object), or are updated only on a clock edge, i.e., it is a synchronous circuit (a *Synchronous* object), or if it is a structural circuit (*Structural* object), i.e., one that is just a set of existing synchronous or combinational circuit elements interconnected together. In each case, the designer defines a new class that inherits from the appropriate class and implements the desired functionality in the constructor and other methods. Individual circuits are interconnected by instantiating *Wire* objects and passing these to the object as arguments to the object constructors.

Software Simulation under JHDL

The actual behavior of the newly defined circuit class is specified differently, depending upon whether it is a *CL* or *Synchronous* object. For *CL* objects, the designer must write a *propagate()* method that will generate a new set of outputs, based on the current inputs, each time it is called. This *propagate()* method is automatically called

by the simulation kernel each time at least one of the input wires connected to the circuit object registers a change during simulation. For *Synchronous* objects, the designer writes a *behavior()* method that will generate a new set of outputs each time it is called. The *behavior()* method is automatically invoked each time a new clock cycle is issued by the simulation kernel. When designers are implementing structural designs (as will be the case when a design library and netlisting capability is available), they only need to derive a new class that inherits from *Structural*, and write a constructor that will wire up the necessary library elements to achieve the desired function. No *propagate()* or *behavior()* methods are written for *Structural* circuits as their behavior is completely derived from the behaviors of the interconnected constituent subcircuits. Any circuit organization is possible and an arbitrary number of hierarchical levels are supported by JHDL.

Currently, the simulation kernel is limited to synchronous, globally clocked circuits but as these are the only kinds of circuits that consistently work on CCMs, this is not a serious limitation¹. However, if multiple clocks are necessary, the simulation kernel can be easily modified to support an arbitrary number of clocks. The JHDL simulator was designed to handle circuits that are run-time re-configured; at any point in the simulation run the clock can be stopped, circuit elements added to or deleted from existing circuitry dynamically, and the simulation run continued from where it stopped. This allows JHDL to simulate complete and partial configuration of circuits on CCMs.

The HWSystem Class

In order to access the JHDL simulation kernel and CCM control layer, the user circuit is encapsulated in a top-level class called *HWSystem*, as shown in Figure 1. The *HWSystem* provides all of the functionality for simulation and CCM control. In addition, the *HWSystem* provides a means of communicating with the external world or CCM host over special wires, called “ports.” These ports synchronize the input and output data with the global clock used in JHDL. The user circuit itself may be composed of an arbitrary number of JHDL circuit objects; only the top-level object is encapsulated by the *HWSystem*.

The *HWSystem* is the essential link between hardware execution and software simulation. It implements a simulation kernel which invokes behavioral descriptions of circuit objects during software simulation. The *HWSystem* is also what implements the API to the CCM device drivers; with this interface, it can coordinate the computation on the CCM by configuring the appropriate circuits on the CCM, loading data, etc. Similarly, the *InPort* or *OutPort* shown

¹some would argue that this limitation is a strength.

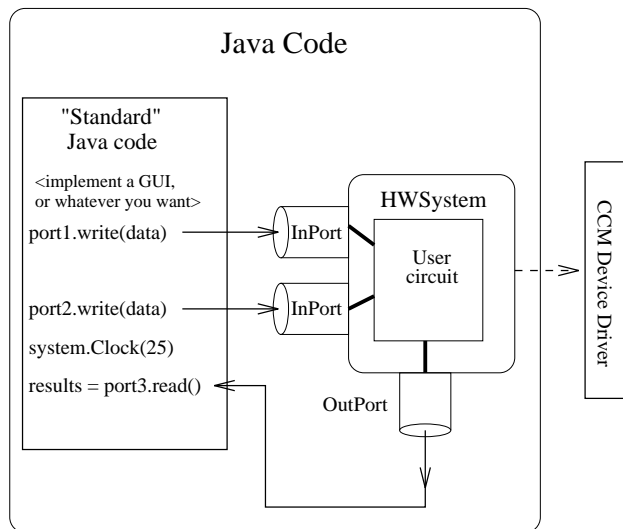


Figure 1: Top level view of a JHDL design

in the figure can either be simulated behaviorally, as software buffers that are synchronized to a global clock, or executed on the CCM, as actual hardware ports that communicate with a host or other hardware. This is shown in Figure 2. Because of this duality, any circuit that can be described in JHDL and compiled to a bitstream can be run interchangeably in the simulation kernel and on a hardware platform, without any modification of the source code. This hardware-software abstraction is discussed further in Section 6.

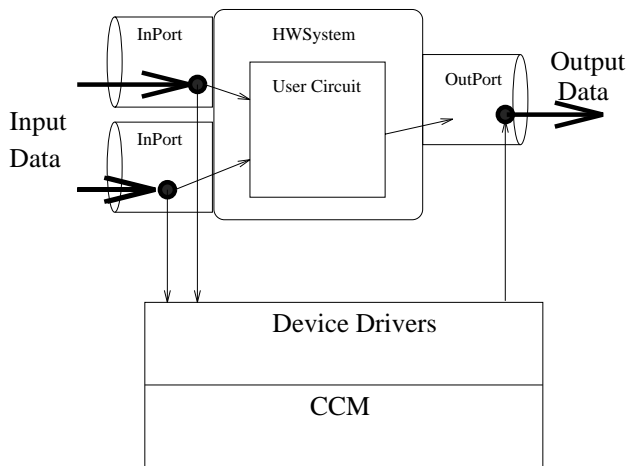


Figure 2: Top level view of JHDL system

Finally, once the circuit is described, the JHDL system is designed to interface to back-end synthesis tools with

relative ease. When a JHDL object is constructed, JHDL automatically maintains an internal circuit graph describing the circuit, including hierarchy and port types (including width and direction). This was designed so that either flat or hierarchical netlists could be easily generated from the internal graph. In addition, this graph is also used by the simulator to perform run-time checks of circuits during simulation. It is also modified to be consistent as circuit elements are added or deleted as when CCM devices are partially or run-time reconfigured. As is the case with all CAD tools, a library of circuit primitives must be provided but once this is available and a netlist format determined, the netlist for any structural circuit will be automatically generated for any structural circuit design. This is discussed further in the future work section.

The Simulation Kernel

The *HWSystem* object tracks all the wire and logic objects that it encapsulates. All input and output data to the top-level circuit is passed via the *InPort* and *OutPort* classes, respectively. These classes provide support for buffered transfers of data. When each circuit object is constructed, it registers itself on a wire list or clock fanout list in the *HWSystem*. This allows the system to track all the pointers needed to perform simulation. The *HWSystem* performs a circuit simulation using the following simple algorithm (assuming that the user has already initialized all data buffers):

1. Cause each *InPort* to drive its wires with the next data in its buffer.
2. Issue a “clock” to all synchronous logic (by calling the *behavior()* method for every *Synchronous* circuit object).
3. Propagate all wires that were updated by the synchronous logic.
4. Propagate all affected combinational logic (by invoking the *propagate()* methods of all *CL* objects in the wires’ fan-outs).
5. Repeat 3-4 above until the network is stable.
6. Cause each *OutPort* to write the state of its associated wire into its buffer.
7. Repeat 1-6 above until the requested number of clocks have been issued.

This algorithm is represented in Figure 3.

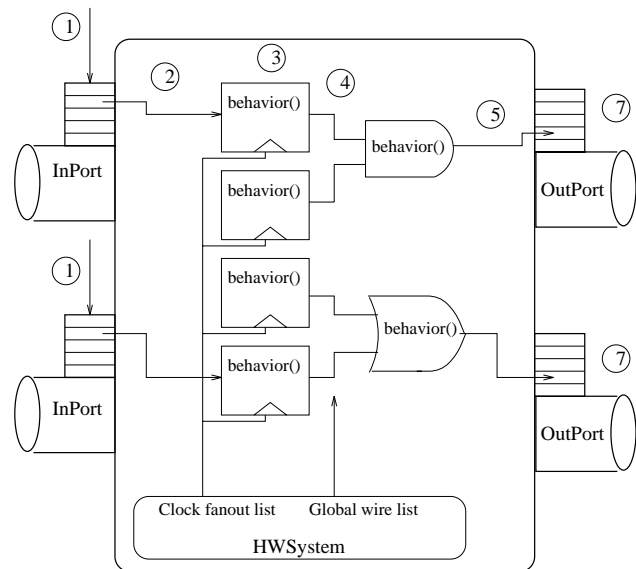


Figure 3: Operation of the simulation kernel

Hardware Execution under JHDL

If the computation is to be performed in the FPGA hardware, the computational steps change significantly. When the *HWSystem* is constructed, it calls the device driver to load in the initial bitstream to the device. Currently, the user must provide the *HWSystem* with the name of the file containing the configuration bitstream. In the near future when netlisting capability is implemented, this will be handled automatically by JHDL. When the *HWSystem* receives the request to clock the circuit, it makes another device driver call to clock the device the requested number of times. The *InPort* and *OutPort* objects make their own driver calls to exchange input/output values with the device. The hardware execution cycle is as follows:

1. The user passes input data to the *InPort* buffer. The *InPort* sends the data to the device via a driver call, and the driver buffers the data.
2. The user requests a sequence of clocks. The *HWSystem* makes a driver call to issue the clocks.
3. The driver issues the clocks and buffers the data for each *OutPort*.
4. The *HWSystem* waits for the driver call to complete. When it does, it requests the data for each *OutPort*, and loads the data into the appropriate objects.

This algorithm is represented in Figure 4.

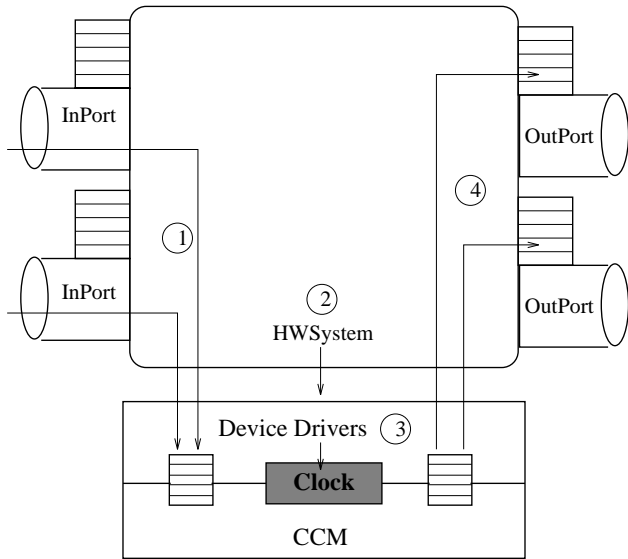


Figure 4: Operation of the hardware interface

Any hardware device can be used that supports the API of the JHDL system. The driver is simply compiled to native code, and the driver calls are linked to the JHDL system as native methods. The driver is loaded at run-time as a shared library; changing the hardware platform is simply a matter of loading a different library file. We implemented a simple JHDL-compatible driver for the VCC HotWorks board, which is based on a Xilinx XC6216-series FPGA. The driver is a simple wrapper around a preexistent device driver which had been developed at BYU, which adds buffering capability and exports the proper API. Further consideration is underway for developing a similar driver for other systems based on Xilinx XC4000-series parts.

7 Modeling Hardware Reconfiguration

In JHDL, circuits are configured and reconfigured on CCMs via object constructors/destructors. However, Java directly supports only explicit object construction; explicit object deletion is not supported as all memory is reclaimed automatically by a run-time garbage collector. In order to support explicit object deletion, JHDL provides a `delete()` method for each of the base circuit classes. When `delete()` is invoked on an object, say object A, A and all of its constituent objects and wires are removed from the internal netlist graph (dereferenced so they can be garbage collected) and marked as deleted. During hardware execution, invoking `delete()` is a signal to the CCM that the device currently occupied by the given circuit should be reclaimed by “blanking” out the appropriate locations in the proper de-

vice.

JHDL supports partial reconfiguration similar to the reconfiguration approach already mentioned through an additional interface class, the *PRSocket* (Partially Reconfigurable Socket). This class is used to describe parts of the circuit that require partial configuration. It maintains the multiple configuration information and automatically provides the transparent switching between simulation and hardware execution. The mnemonic implies that partial reconfiguration is modeled as a discrete chip socket that allows any chip with the right pinout to be “plugged in.” The socket simply serves as a placeholder in the internal JHDL netlist. The user connects the static logic in the circuit to this socket, and provides the behavioral description of what goes on inside that socket later. The *PRSocket* will allow any circuit with the proper interface to be “plugged in”. The basic function of the *PRSocket* is illustrated in Figure 5. When the computation is being performed in the simulator, the *PRSocket* simply dereferences all pointers to the underlying JHDL object and creates an instance of the newly configured object. When the computation is performed on the hardware platform, the *PRSocket* communicates with the hardware drivers to load the new partial configuration at the appropriate chip location.

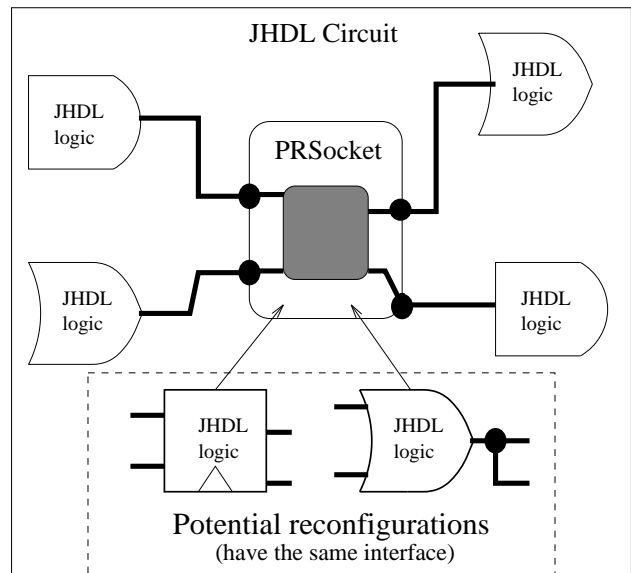


Figure 5: Describing partial reconfiguration

Partial Reconfiguration: Software Simulation

The user needs to tell the *PRSocket* in advance which configurations it will encapsulate. This constraint helps the *PRSocket* develop a netlist, and is also needed to facilitate

hardware execution. The list of configurations is encapsulated in an object called a *ConfigGroup* by defining a function that might look like this:

```
class myConfigGroup extends ConfigGroup {
...
/* A Node is the base class for all JHDL logic
Node getNewCircuit(int id, PRSocket sock) {
  switch(id) {
  case 1:
    return new Circuit1(...);
  case 2:
    return new Circuit2(...);
  case 3:
    return new Circuit3(...);
  ...
  } } }
```

This *ConfigGroup* object is passed to the *PRSocket* when it is constructed. Once the *PRSocket* is constructed, the user connects it to the static logic just like any other piece of logic. During circuit construction it serves as a placeholder in the JHDL internal netlist.

When the user wants to reconfigure the circuit, he tells the *PRSocket* to load configuration #*n* by calling *PRSocket.Reconfigure(n)*. The *PRSocket* then calls *ConfigGroup.getNewCircuit(n)* to get an instance of the desired circuit object. The new object gets pointers to the static interface wires from the *PRSocket*, and adds itself to the global netlist as usual. It can then be controlled by the simulation kernel.

To complete the reconfiguration process, the *PRSocket* must destroy the old logic that it contained. This is performed by invoking the *delete()* method on the circuit to be deleted. Before simulating again, the kernel sweeps its global object lists and removes all references to deleted objects. The partial reconfiguration process is depicted in Figure 6. A more detailed look at the how to write JHDL code for partial reconfiguration is given in Section 8.

7.1 Partial Reconfiguration: CCM Execution

The *HWSystem* keeps track of all the partial configuration lists that are used in its *PRSockets*. When it is constructed, it instructs the CCM device driver to keep a pointer to all partial configurations, as well as loading the static configuration. When the *Reconfigure()* method of a *PRSocket* is called, it references the partial configuration by an ID number and calls the device driver to load the appropriate configuration into the corresponding part of the circuit. This of course requires a hardware platform that can perform partial configuration, like the Xilinx XC6200 series. For this project, the BYU device driver for the VCC Hotworks board has been augmented to support partial configuration.

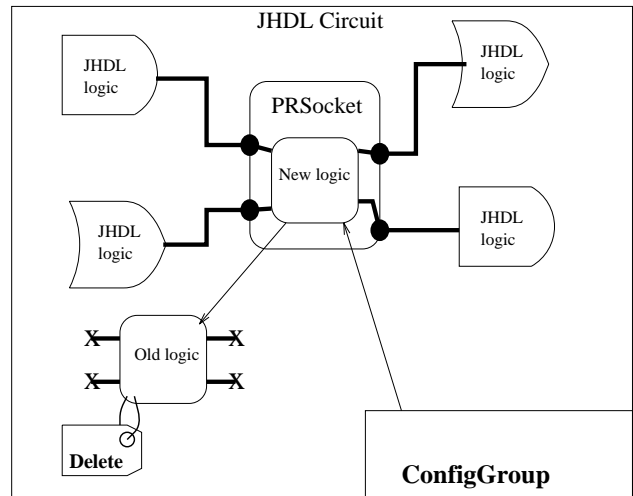


Figure 6: Partial reconfiguration during simulation

8 JHDL Examples

JHDL has already been used to describe and execute several “real” applications, such as the “shapesum” and “correlation” functions of the Chunky-SLD Automatic Target Recognition (ATR) problem [8]. These applications have been implemented on the Xilinx 6200 using partial reconfiguration, as well as on other platforms [10]. For this initial feasibility study, the original circuits are being used as they were originally implemented via schematic capture and manual placement. The main difference is that the entire circuit is now described in JHDL. This description provides a comprehensive simulation model of the ATR application. This is itself noteworthy as the ATR application consists of several partial configurations that are loaded into FPGA devices as new images are correlated [10]. In addition, this *same* JHDL description is used to directly control the VCC “Hotworks” CCM, replacing the original Tcl/Tk program that was previously used for controlling the CCM. Because netlisting capability is not yet present in JHDL, the correspondence between JHDL circuit object and configuration bitstream is managed manually by the user through the *ConfigGroup* as already discussed². Even in these early stages, JHDL has proven to be surprisingly effective at describing, simulating, and executing these systems, including the partial reconfiguration required to change the image template.

Although the ATR examples are currently operational, they are far too complex to serve as coding examples of

²In practice this means that the designer is responsible for generating bitstreams with some tool and “telling” JHDL where these files are and what circuit objects they are associated with. In the near future much of this will be completely automated.

JHDL in this paper. As a more accessible example of JHDL usage, consider the following FIR filter example that will be used to demonstrate the salient features of the JHDL system. For this example, assume that the tap weights are compiled directly into the circuit and partial re-configuration is used to modify the weights at runtime. At the top level, the user would write standard Java to provide the user interface, gather input data, and so forth. Assuming that the FIR filter has already been designed (the next section illustrates the FIR design), it would be “wired” into the top-level system as shown below.

```
class myJavaCode {
  SomeMethod(...) {
    /* Create a new system */
    HWSystem system = new HWSystem();

    /* Tell the system how to compute -- this time, we'll simulate */
    system.setSWMode();

    /* Create new wires to pass to the filter, 8 bits wide each.
    The "system" reference helps the Wire class build the circuit
    graph ("system" is the parent) */
    Wire Input = new Wire(system, 8);
    Wire Output = new Wire(system, 8);

    /* Create a new filter object; pass in pointers to I/O wires.
    Note that this configures when in hardware mode. */
    FirFilter filter = new FirFilter(system, Input, Output);

    /* Encapsulate the I/O wires with ports. */
    InPort inport = system.newInPort(Input);
    OutPort outport = system.newOutPort(Output);

    /* The object is now constructed and appropriately
    encapsulated. Now, reconfigure the tap constants; this method
    is user-defined. */
    filter.Reconfigure(GetNewTapConstants());

    /* Now, initialize the input buffer with data. */
    inport.writeBuffer(InputData);
    /* Allocate a new output buffer, same size as the data array */
    outport.newBuffer(InputData.length);

    /* Now clock the circuit, and get the results. */
    system.Clock(some_number);
    int Results[] = outport.getBuffer();
    ...
  }
}
```

The user can then take the results and process them as needed. Now, let’s look at how the FIR filter circuit might be described.

```
/* This is just a structural circuit - all behavior is de-
scribed in the fir taps */
class FirFilter extends Structural {
  Wire[] data_wire_array, mac_wire_array;
```

```
Wire fir_zero, data_input, data_output, fir_output;
PRSocket FirTaps[];
int tapCount;

/* This manages the partial reconfigurations for each fir tap */
static FirConfigGroup config = new FirConfigGroup();

FirFilter(Node parent, Wire in, Wire out) {
  /* Every object that inherits from Node must do this first to
  build the netlist graph */
  super(parent);

  /* Now, declare my inputs/outputs, 8 bits each */
  inPort(in, 8); outPort(out, 8);

  fir_zero = new Wire(this, 8);
  ... { initialize other wires in a similar manner }

  for(i=1;i<TapCount;i++) {
    FirTaps = new PRSocket(this, config);
    /* Now we must declare the static interface to each PRSocket.
    Each wire is assigned a "port number" for reference. */
    FirTaps[i].inPort(data_wire_array[i-1], 0);
    FirTaps[i].inPort(mac_wire_array[i-1], 1);
    FirTaps[i].outPort(data_wire_array[i], 2);
    FirTaps[i].outPort(mac_wire_array[i], 3);
  }
}

/* The user defines this to export a top-level Reconfigure
method to the outside world. However, the actual
reconfiguration of each individual circuit is handled by the
PRSocket. */
public void Reconfigure(int tap_constants[]) {
  for (i=0;i<TapCount;i++) {
    FirTaps[i].Reconfigure(tap_constants[i]);
  }
}
}
```

The user of course has to define the *FirConfigGroup* so that it returns the kind of object he wants. This could be done as follows:

```
class FirConfigGroup extends ConfigGroup {
  /* Tell the superclass how many ports (4) and potential
  configurations (8 bit fir tap constant  $\implies$  256) are allowed. */
  public FirConfigGroup(HWSystem s) { super(s, 4, 256); }

  /* Here, the reconfiguration is completely described by
  returning pointers to new objects of the desired type. In our case,
  the id just represents the tap constant and associated name of the
  file containing the configuration to be loaded. */
  public Node getPRObject(int id, PRSocket sock) {
    return new FirTap(sock, id);
  }
}
```

Finally, the user must describe the behavior of the fir tap. The logical choice is to make this a *Synchronous* object. It could be implemented as follows:

```
class FirTap extends Synchronous {
  int tap_constant;
  Wire fir_input, data_input, mac_output, data_output, d0_d1;

  public FirTap(PRSocket p, int constant) {
    int tap_constant;
    Wire d0_d1;

    /* The PRSocket is a Node, so it is the parent object. */
    super(p);

    tap_constant = constant;

    /* Now, get a pointer to all the wires that interface to the static
    logic. The data_input wire was assigned port #0; etc. */
    data_input = p.inConnect(0);
    fir_input = p.inConnect(1);
    data_output = p.outConnect(2);
    fir_output = p.outConnect(3);

    inPort(data_input, 8); ...
  }

  /* Here, we describe the actual computation of each tap.
  This is executed by the HWSYSTEM once per clock. */

  public void behavior() {

    /* Multiply-accumulate the input value, and delay the input
    value. Wire values are read and written using the get() and
    put() methods, respectively. We pass a pointer to "this" with
    every access, which is used to help enforce port directions
    (necessary for netlisting). */

    fir_output.put(this, tap_constant * d0_d1.get(this)
      + fir_input.get(this));
    data_out.put(this, d0_d1.get(this));
    d0_d1.put(this, data_in.get(this));
  }
}
```

9 Evaluation and Conclusions

We believe that the constructor/destructor mechanism has proven to be a feasible way to control configuration on a CCM. In addition, JHDL met all of the project goals that were defined at the outset of this research project:

1. JHDL is based on a popular language and requires no language extensions for circuit design.

2. The CCM control paradigm is CCM independent, adopting the object-instance construction metaphor from object-oriented languages. The abstraction will work with any standard CCM and work is now under way to interface JHDL to other CCMs such as the Wildforce system from Annapolis Microsystems.

3. JHDL supports both partial and global configuration and demonstration applications from ATR have been implemented to show this capability.

4. A JHDL application description serves as both simulation and execution for CCM applications. No code modifications are required and switching between software simulation and hardware execution on the CCM requires the setting of a single boolean variable.

JHDL also provides additional benefits because it is based on a commonly-used programming language and as such all of the standard language features, such as I/O, are accessible to the designer throughout the design process. Unlike VHDL for example, it is quite easy to perform arbitrary I/O in JHDL, both to the console and to files during software simulation. Of course, some of these features are only accessible during simulation mode in JHDL; however, that is when they may be of the most use. For example, designers can easily insert print statements in their code as a debugging aid so that the internal state of the application can be ascertained during a simulation run. Once netlisting is fully implemented, these I/O statements will just be ignored. Graphical User Interfaces (GUIs) can also be easily added to the program without the need for any complex linking; they are just part of the JHDL application as the complete Java API is available to the designer when writing JHDL applications. JHDL has the added advantage that the GUI (or any other software written and integrated with the application) may be run on the host even when the circuit parts of the application are executing on the CCM. This is possible because JHDL allows the application to be divided into those parts that will run in software and those parts that will run in hardware. When operating in hardware execution mode, only those parts of the application that are described using circuit classes will be executed on the CCM platform. All other parts of the application remain on the host, operating essentially as a separate program that is communicating with the user-designed circuitry via the CCM device driver. In this way, JHDL allows for both software and hardware descriptions to not only co-exist but also to coexecute.

10 Future Work

In addition to continuing experimentation with JHDL for CCM applications, two areas have been identified for further work in JHDL: netlisting, to allow JHDL to function as a complete structural design tool, and behavioral synthesis, to allow circuits to be expressed at a higher level. Indeed, behavioral compilation was one of the first ideas discussed in the early proposal phase of this project. However, because it was necessary to first prove feasibility of the basic concept of using object-instance construction as a metaphor for CCM control, further development in this area had to be delayed. In addition, JHDL is showing significant potential as a purely structural design tool and there is now interest at BYU in further developing JHDL in this direction as well. The need for netlisting has already been discussed in this paper and the necessary internal circuit graph has already been fully implemented. What remains is to select a netlist format and develop sets of circuit libraries that are based on today's popular FPGA devices. This effort is already underway.

Behavioral synthesis will be designed to exploit industry CAD tools such as VHDL synthesis. The approach is based on a fourth circuit class, *HWProcess* that is already partially implemented. Similar to the way circuits are defined using the other *CL* and *Structural* classes, the designer inherits from *HWProcess* when behavioral synthesis is desired. The *HWProcess* class provides an additional method, *waitUntilClock()*, that designers insert into their *behavior()* methods. It provides the same basic behavior as a *wait()* in a VHDL process. Designers will be able to express circuits behaviorally with this class using a subset of Java statements and developing a circuit description that uses the common *wait until clock* idiom found in most VHDL synthesizable subsets. The subset of Java statements will be limited to statements that can be supported by synthesizable VHDL subsets. This JHDL code will then be translated to VHDL through a simple syntax-directed textual substitution and processed with VHDL synthesis tools. The advantage of this approach is that it allows the circuit to be simulated in JHDL in a natural context with other circuits but also provides a clear path to behavioral synthesis that leverages currently available tools.

References

- [1] J. Burns, A. Donlin, J. Hogg, S. Singh, and M de Wit. A dynamic reconfiguration run-time system. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 66–75, Napa, CA, april 1997.
- [2] S. Gehring and S. Ludwig. The trianus system and its application to custom computing. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, pages 176–184, Darmstadt, Germany, September 1996. Springer-Verlag.
- [3] M. Gokhale and E. Gomersoll. High level compilation for fine grained fpgas. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 165–173, Napa, CA, April 1997.
- [4] C. Iseli and E. Sanchez. Spyder: A reconfigurable VLIW processor using FPGAs. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 17–24, Napa, CA, April 1993.
- [5] W. Luk, N. Shirazi, and P. Y. K. Cheung. Compilation tools for run-time reconfigurable design. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 56–65, Napa, CA, April 1997.
- [6] P. Lysaght and J. Stockwood. A simulation tool for dynamically reconfigurable field programmable gate arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(3):381–390, September 1996.
- [7] L. Moll and M. Shand. Systems performance measurement on PCI pamette. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 125–133, Napa, CA, April 1997.
- [8] M. Rencher and B. Hutchings. Automated target recognition on splash 2. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 192–200, Napa, CA, April 1997.
- [9] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.
- [10] M. J. Wirthlin and B. L. Hutchings. Improving functional density through run-time constant propagation. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 86–92, Monterey, CA, February 1997.